(2)

AVF Control Number: AVF-VSR-90502/41

Ada* COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 880623N1.09133
SD-SCICON plc
SD VAX/VMS x MC68020 Ada-Plus,3B.00
Local Area VAX Cluster x Motorola MC68020

Completion of On-site Testing:
23 June 1988

Prepared By:
The National Computing Centre Limited
Oxford Road
Manchester M1 7ED
United Kingdom

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

* Ada is a registered trademark of the United States Government (Ada Joint Program Office).

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETETEING FORM

1. REPORT NUMBER | 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) Ada Compiler Validation Summary Report: SD-SCICON plc, SD VAX/VMS x MC68020 Ada-Plus, 3B.00, Local Area VAX Cluster (Host) to Motorola MC68020 (Target). (880623N1.09133)	5. TYPE OF REPORT & PERIOD COVERED 23 June 1988 to 23 June 1989 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(*s*) National Computing Centre, Ltd. Manchester, UK.	8. CONTRACT OR GRANT NUMBER(*s*)
9. PERFORMING ORGANIZATION AND ADDRESS National Computing Centre, Ltd. Manchester, UK.	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE 23 June 1988 13. NUMBER OF PAGES 52 p.
14. MONITORING AGENCY NAME & ADDRESS(*If different from Controlling Office*) National Computing Centre, Ltd. Manchester, UK.	15. SECURITY CLASS (*of this report*) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (*of this Report*)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20. If different from Report*)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (*Continue on reverse side if necessary and identify by block number*)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

SD VAX/VMS x MC68020 Ada-Plus, 3B.00, SD-SCICON plc, National Computing Centre, Ltd., Local Area VAX Cluster under VMS, V4.6 (Host) to Motorola MC68020 (MVME 133) No operating system (Target), ACVC 1.9.

Ada* Compiler Validation Summary Report:

Compiler Name: SD VAX/VMS x MC68020 Ada Plus, 3B.00

Certificate Number: 880623N1.09133

Host:
Digital Equipment VAX Cluster comprising VAX 8600 seven MicroVAX IIs and VAX workstation 2, under VMS, V4.6

Target:
Motorola MC68020 implememted on MVME 133 Board
No operating system

Testing Completed 23 June 1988 Using ACVC 1.9

This report has been reviewed and is approved.

AEJ. Pink

The National Computing Centre Ltd
Jane Pink
Oxford Road
Manchester, M1 7ED
United Kingdom

John F Kramer

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Virginia L. Castor

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20

* Ada is a registered trademark of the United States Government (Ada Joint Program Office).

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report ~~(VSR)~~ describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada Compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behaviour that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:-

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard

- To determine that the implementation-dependent behaviour is allowed by the Ada Standard.

Testing of this compiler was conducted by NCC under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 23 June 1988 at SD-SCICON plc, Pembroke House, Pembroke Broadway, Camberley, Surrey.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:-

The National Computing Centre Ltd
Oxford Road
Manchester M1 7ED
United Kingdom

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:-

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles sucessfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support are self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: SD VAX/VMS x MC68020 Ada Plus, 3B.00

ACVC Version: 1.9

Certificate Number: 880623N1.09133

Host Computer:

Machine: Local Area VAX cluster comprising VAX 8600, seven microVAX IIs and VAX workstation 2

Operating System: VMS V4.6

Memory Size: 83Mb

Target Computer:

Machine: Motorola MC68020 implemented on MVME/133 board

Operating System: no operating system

Memory Size: 1Mb

Communications Network: RS232

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behaviour of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation supports the additional predefined types SHORT_INTEGER, LONG_INTEGER, and LONG_FLOAT, in the package STANDARD. (See tests B86001C and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

  Apparently no default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

  Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Apparently NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Sometimes NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is gradual. (See tests C45524A..Z.)

. Rounding.

The method used for rounding to integer is apparently round to even.(see tests C46012A..Z.).

The method used for rounding to longest integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception (See test C36003A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises no exception. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications . (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

. Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Record representation clauses are not supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are not supported. (See test C87B62A.)

- Pragmas.

The pragma INLINE is not supported for procedures. The pragma INLINE is not supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

The Director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behaviour for SEQUENTIAL_10, DIRECT_10 and TEXT_10.

. Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 27 tests had been withdrawn because of test errors. The AVF determined that 400 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation and 206 executable tests that use file operations not supported by the implementation. Modifications to the code, processing, or grading for 10 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	108	1050	1465	17	11	44	2695
Inapplicable	2	1	388	0	7	2	400
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	190	499	548	248	166	98	141	326	131	36	234	3	75	2695
Inapplicable	14	73	126	0	0	0	2	1	6	0	0	0	178	400
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	C35904A	C37215C	C41402A	CC1311B
	C35904B		C45332A	
E28005C	C35A03E	C37215E	C45614C	BC3105A
C34004A	C35A03R	C37215G	A74016C	AD1A01A
C35502P	C37213H	C37215H	C85018B	CE2401H
A35902C	C37213J	C38102C	C87B04B	CE3208A

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 400 tests were inapplicable for the reasons indicated:

- C35702A uses SHORT_FLOAT which is not supported by this implementation.

- A39005G uses a record representation clause. Although record representation clauses are supported, there are restrictions. The applicable restriction for this test is on the alignment clause- see section F.4.2.1 of Appendix F.

- C45231D requires a macro substitution for any predefined numeric types other than INTEGER, SHORT_INTEGER, LONG_INTEGER, FLOAT, SHORT_FLOAT, and LONG_FLOAT. This compiler does not support any such types.

- C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

- C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

- C4A013B uses a static value that is outside the range ofthe most accurate floating-point base type. The test executes and reports NOT_APPLICABLE.

- B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

- C86001F redefines the package SYSTEM, but TEST_IO (a package used to collect the executable test results) is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependant on the package TEST_IO.

- C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.

- CA3004E, EA3004C, and LA3004A use the INLINE pragma for procedures, which is not supported by this compiler.

- CA3004F, EA3004D, and LA3004B use the INLINE pragma for functions, which is not supported by this compiler.

- The following 178 tests are inapplicable because sequential, and direct access files are not supported.

CE2102C	CE2102G..H(2)	CE2102K	CE2104A..D(4)
CE2105A..B(2)	CE2106A..B(2)	CE2107A..I(9)	CE2108A..D(4)
CE2109A..C(3)	CE2110A..C(3)	CE2111A..E(5)	CE2111G..H(2)
CE2115A..B(2)	CE2201A..C(3)	CE2201F..G(2)	EE2201D..E(2)
CE2204A..B(2	CE2208B	CE2210A	

TEST INFORMATION

			EE2401D
CE2401A..C(3)	CE2401E..F(2)	CE2404A	EE2401G
CE2405B	CE2406A	CE2407A	CE2408A
CE2409A	CE2410A	CE2411A	AE3101A
CE3102B	EE3102C	CE3103A	CE3104A
CE3107A	CE3108A..B(2)	CE3109A	CE3110A
CE3111A..E(5)	CE3112A..B(2)	CE3114A..B(2)	CE3115A
CE3203A		CE3301A..C(3)	CE3302A
CE3305A	CE3402A..D(4)	CE3403A..C(3)	CE3403E..F(2)
CE3404A..C(3)	CE3405A..D(4)	CE3406A..D(4)	CE3407A..C(3)
CE3408A..C(3)	CE3409A	CE3409C..F(4)	CE3410A
CE3410C..F(4)	CE3411A	CE3412A	CE3413A
CE3413C	CE3602A..D(4)	CE3603A	CE3604A
CE3605A..E(5)	CE3606A..B(2)	CE3704A..B(2)	CE3704D..F(3)
CE3704M..O(3)	CE3706D	CE3706F	CE3804A..E(5)
CE3804G	CE3804I	CE3804K	CE3804M
CE3805A..B(2)	CE3806A	CE3806D..E(2)	CE3905A..C(3)
CE3905L	CE3906A..C(3)	CE3906E..F(2)	

Results of running a subset of these tests showed that the proper exceptions are raised for unsupported file operations.

- The following 201 tests require a floating-point accuracy that exceeds the maximum of 15 digits supported by this implementation:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behaviour. Modifications are made by the AVF in cases where legitimate implementation behaviour prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behaviour that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for only 4 class C tests and 6 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A B29001A B91001H BC2001D BC2001E BC3204B

- C4A012B This test checks that 0.0 raised to a negative value raises CONSTRAINT_ERROR; however, NUMERIC_ERROR is also an acceptable exception to be raised in this case. Thus, conforming implementations must either "pass" this test or print failure messages that indicate that the "WRONG EXCEPTION" was raised.

- C64104M, CB1010B, C95085M, modified versions using representation clauses to increase the collection sizes for C64104M and CB1010B and C95085M to 8K Bytes, 4K Bytes and 4K Bytes respectively, were prepared. These modified tests executed susccessfully. The compiler will also allow the default collection size to be altered using a compiler option, this facility was tested and resulted in tests which executed successfully.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the SD VAX/VMS x MC68020 Ada-Plus, 3B.00 was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behaviour on all inapplicable tests.

3.7.2 Test Method

Testing of the SD VAX/VMS x MC68020 Ada-Plus, 3B.00 using Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a Local Area VAX Cluster host operating under VMS, V4.6, and a Motorola MC68020 target with no operating system. The host and target computers were linked via RS232.

A magnetic tape containing all tests was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the pre-validation testing were not included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked on the Local Area VAX Cluster, and all executable tests were run on the Motorola MC68020. Object files were linked on the host computer, and executable images were transferred to the target computer via RS232. Results were printed from the host computer, with results being transferred to the host computer via RS232.

The compiler was tested using command scripts provided by SD-SCICON plc and reviewed by the validation team. The compiler was tested using all default option settings.

Tests were compiled, linked, and executed (as appropriate) using a Local Area VAX Cluster comprising a VAX 8600, seven Microvax IIs and a VAX workstation 2 connected via ethernet, as the host computers and a single target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at SD-SCICON plc, Pembroke House, Pembroke Broadway, Camberley and was completed on 23 June 1988.

APPENDIX A

DECLARATION OF CONFORMANCE

SD-SCICON plc have submitted the following Declaration of Conformance concerning the SD Motorola VAX/VMS x MC68020 Ada Plus, 3B.00

DECLARATION OF CONFORMANCE

Compiler Implementor : SD-SCICON plc

Ada* Validation Facility : The National Computing Centre Limited, Oxford Rd, Manchester, M17ED

Ada Compiler Validation Capability (ACVC) Version: 1.9

BASE CONFIGURATION

Base Compiler Name	:	SD VAX/VMS x MC68020 Ada-Plus
Version	:	3B.00
Host Architecture	:	DEC Local Area VAX Cluster comprising of a VAX 8600, seven MicroVAX IIs and VAX Workstation 2.
Host Operating System	:	VMS
Version	:	V4.6
Target Architecture	:	Motorola MC68020 implemented on MVME 133 board
Target Operating System	:	No operating system

I, the undersigned, representing SD-SCICON plc, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that SD-SCICON plc is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

R Barnett Date 4-AUG-88

Name of Person signing R.E. BARNETT

Title : Customer Services Production Group Manager

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

Owner's Declaration

I, the undersigned, representing SD-SCICON plc, take full responsibility for the implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

R. Barnett

Date: 4-AUG-88

Name of Person signing : R.E BARNETT

Title : Customer Services Production Group Manager

Name of Base Compiler Owner :

SD-Scicon plc

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the SD VAX/VMS x Motorola MC68020 implememted on MVME board Ada Plus, are described in the following sections, which discuss topics in Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

Systems Designers

Ada-Plus
VAX/VMS
MC68020
Appendix F to the Reference Manual

Reference:	D.A.REF.AF[BC-MH]
Issue:	3.0
Date of Issue:	February 1988
Software Version:	Release 3A.00

Systems Designers Software Technology, Camberley, Surrey, U.K. †

Systems Designers Software Inc., Cambridge, Massachusetts, U.S.A.

† Systems Designers plc registered in England 1642767

February, 1988

This document was prepared using VAX DOCUMENT, Version 1.0

TABLE OF CONTENTS

FIGURES

Preface

This document describes the implementation-dependent characteristics of the Ada compiler supplied with VAX/VMS x MC68020 Ada-Plus.

The document should be considered to be Appendix F to ANSI/MIL-STD-1815A-1983, *Reference Manual for the Ada Programming Language [LRM]*.

References

{ALRM}	Ada-Plus VAX/VMS MC68020 Assembly Language Reference Manual: D.A.REF.ALRM[BC-MH]
{TH}	Ada-Plus VAX/VMS MC68020 Target Handbook: D.A.REF.TH[BC-MH]
[LRM]	Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, US Department of Defense, 22 January 1983

APPENDIX F

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

F.1 Implementation-Dependent Pragmas

F.1.1 Pragma DEBUG

Form

pragma DEBUG ([NAME =>] value) ;

The pragma takes a single argument *value* which is the name of a scalar or access type.

Position

The pragma can be placed at the position of a basic_declarative_item, a later_declarative_item or a statement.

Effect

The effect of the pragma is to cause the compiler to generate out-of-line code that writes the *value* to a buffer. The code is optionally executed by the Debug System by inserting a breakpoint at the position of the pragma in the code.

F.1.2 Pragma EXPORT

Form

pragma EXPORT ([ADA_NAME =>] name, [EXT_NAME =>] name_string) ;

The pragma takes two arguments, *name* and *name_string*. The *name* must be the simple name of a variable at the package level and *name_string* must be a string literal that is unique for the entire program.

Position

The pragma can be placed at the position of a basic_declarative_item of a library_package_specification or in the declarative_part of a library_package_body.

Effect

The effect of the pragma is to cause the compiler to generate additional builder information that associates the *name* with the *name_string*. This external naming is restricted to static data objects.

The parameter *name_string* must conform to the naming conventions imposed by the MC68020 Assembler, see *Assembly Language Reference Manual* {*ALRM*} for details.

F.1.3 Pragma SQUEEZE

Form

pragma SQUEEZE (*type*_simple_name) ;

Takes the simple name of record or array type as a single argument.

Position

The allowed positions for this pragma, and the restrictions on the named type, are governed by the same rules as for a representation clause; in particular, the pragma must appear before any use of representation attribute of the squeezed entity.

Effect

The pragma specifies that storage minimization to bit level is to be used as the main criterion when selecting the representation of the given type.

F.1.4 Pragma SUPPRESS_ALL

Form

pragma SUPPRESS_ALL ;

Position

The pragma must occur before anything else in the source file apart from comments or other pragmas.

Effect

The effect of the pragma is to request that the compiler leaves out all run-time checks for CONSTRAINT_ERROR and NUMERIC_ERROR.

F.1.5 Pragma SUPPRESS_STACK

Form

pragma SUPPRESS_STACK ;

Position

The pragma must occur before anything else in the source file apart from comments or other pragmas.

Effect

The effect of the pragma is to request that the compiler leaves out all run-time checks for STORAGE_ERROR.

F.2 Implementation-Dependent Attributes

There are no such attributes.

F.3 Package SYSTEM

The specification of the package SYSTEM is given in Figure F-1.

Figure F-1: Package SYSTEM

```
package SYSTEM is

   type ADDRESS is private;

   type NAME is (MC68020);

   SYSTEM_NAME  : constant NAME := MC68020;
   STORAGE_UNIT : constant      := 8;
   MEMORY_SIZE  : constant      := 2**32;

   MIN_INT      : constant      := -(2**31);
   MAX_INT      : constant      := (2**31)-1;

   MAX_DIGITS   : constant      := 15;

   MAX_MANTISSA : constant      := 31;

   FINE_DELTA   : constant      := 2#1.0#E-31;
   TICK         : constant      := 2#1.0#E-7;


   subtype PRIORITY is INTEGER range 0 .. 126;

   type    UNIVERSAL_INTEGER is range MIN_INT .. MAX_INT;

   subtype EXTERNAL_ADDRESS is STRING;


   function  CONVERT_ADDRESS (ADDR   : EXTERNAL_ADDRESS)
      return ADDRESS;

   function  CONVERT_ADDRESS (ADDR   : ADDRESS)
      return EXTERNAL_ADDRESS;

   function  "+"             (ADDR   : ADDRESS;
                              OFFSET : UNIVERSAL_INTEGER)
      return ADDRESS;

private

   -- Implementation-dependent type ADDRESS

end SYSTEM;
```

F.3.1 Function CONVERT_ADDRESS

In order to obtain addresses, the function CONVERT_ADDRESS is supplied. The function takes a parameter of type EXTERNAL_ADDRESS which must be eight or less hexadecimal characters representing an address. If the address is outside the range of 0..MEMORY_SIZE, the predefined exception CONSTRAINT_ERROR is raised. CONSTRAINT_ERROR is also raised if the EXTERNAL_ADDRESS contains any non-hexadecimal characters.

The function is overloaded to take a parameter of type ADDRESS and return an EXTERNAL_ADDRESS. This value has all leading zeros suppressed unless the address is zero, in which case a single zero is returned.

Examples:

```
ADDR     := CONVERT_ADDRESS ("0C45");      -- address 3141
STR      := CONVERT_ADDRESS (ADDR);        -- STR(1..3) = "4C45"
VAR      := CONVERT_ADDRESS (VARIABLE'ADDRESS);
```

F.4 Restrictions on Representation Clauses

F.4.1 Length Clauses

F.4.1.1 Attribute SIZE

The value specified for SIZE must not be less than the minimum number of bits required to represent all values in the range of the associated type or subtype. Otherwise, a Compiler Restriction is reported.

F.4.1.2 Attribute SMALL

There are no restrictions for this attribute.

F.4.1.3 Attribute STORAGE_SIZE

For access types the limit is governed by the address range of the target machine and the maximum value is determined by SYSTEM.ADDRESS'LAST.

For task types the limit is also SYSTEM.ADDRESS'LAST.

F.4.2 Record Representation Clauses

F.4.2.1 Alignment Clause

The static_simple_expression used to align records onto storage unit boundaries must deliver the values 0 (bit aligned), 1 (byte aligned), 2 (word aligned) or 4 (long word aligned).

F.4.2.2 Component Clause

Non-scalar types must be aligned and sized correctly.

The component size defined by the static range must not be less than the *minimum* number of bits required to hold every allowable value of the component. For a component of non-scalar type, the size must not be larger than that chosen by the compiler for the type.

F.4.3 Address Clauses

Address clauses are implemented as assignments of the address expressions to objects of an appropriate access type.

An object being given an address is assumed to provide a means of accessing memory external to the Ada program. An object declaration with an address clause is treated by the compiler as an access object whose access type is the same as the type of the object declaration. This access object is initialised with the given address at the point of elaboration of the corresponding address clause. For example, consider:

```
X : INTEGER;
  .
  .
  .

for X'ADDRESS use at CONVERT_ADDRESS("FF00");
```

This is equivalent to:

```
type X_P is access INTEGER;
X : X_P;
    .
    .
    .
X := new_AT_ADDRESS(X_P, "FF00");
--  where  function  new_AT_ADDRESS  claims no  store  but returns the address given.
```

NOTE

See Section F.6 for interpretation of expressions in address clauses and Section F.3.1 for information on CONVERT_ADDRESS.

It is the responsibility of some external agent to initialise the area at a given address. The Ada program may fail unpredictably if the storage area is initialised prior to the elaboration of the address clause. The access object can be used for reading from and writing to the memory normally, but only after the elaboration of the address clause.

Address clauses can only be given for objects and task entries. Address clauses are not supported for other entities.

Unchecked Storage Deallocation will not work for objects with address clauses.

F.4.3.1 Object Addresses

For objects with an address clause, a pointer is declared that points to the object at the given address. There is a restriction, however, that the object cannot be initialised either explicitly or implicitly (i.e. the object cannot be an access type).

F.4.3.2 Subprogram, Package and Task Unit Addresses

Address clauses for subprograms, packages and task units are not supported by this version of the compiler.

F.4.3.3 Entry Addresses

Address clauses for entries are supported; the address given is the address of an interrupt vector. See the *Target Handbook* {*TH*} for details.

Example:

```
task INTERRUPT_HANDLER is
   entry DONE;
   for DONE use at SYSTEM.CONVERT_ADDRESS ("7C");
end INTERRUPT_HANDLER;
```

Note that it is only possible to define an address clause for an entry of a single task.

F.5 Implementation-Generated Names

There are no implementation-generated names denoting implementation-dependent components.

F.6 Interpretation of Expressions in Address Clauses

The expressions in an address clause are interpreted as absolute addresses on the target. Address clauses for subprograms, packages and tasks are not implemented.

F.7 Unchecked Conversions

The implementation imposes the restriction on the use of the generic function UNCHECKED_CONVERSION that the size of the target type must not be less than the size of the source type.

F.8 Characteristics of the Input/Output Packages

The predefined input/output packages SEQUENTIAL_IO, DIRECT_IO and TEXT_IO are implemented as "null" packages that conform to the specification given in the *Ada Language Reference Manual [LRM]*. The packages raise the exceptions specified in *Ada Language Reference Manual [LRM], Chapter 14*. There are four possible exceptions that can be raised by these packages. These are given here in the order in which they will be raised:

a. The exception STATUS_ERROR is raised by an attempt to operate upon a file that is not open, i.e. any files other than the standard input and output files (since no files can be opened).

b. The exception MODE_ERROR is raised if any input operation is attempted using the standard output file or if any output operation is attempted using the standard input file.

c. The exception USE_ERROR is raised upon any attempt to create or open a file, or to set line or page lengths to any value other than UNBOUNDED.

d. The exception END_ERROR is raised if any input operation is attempted from the standard input file. Note that the standard output file is treated as a character sink, and output operations to it have no effect.

Note that NAME_ERROR cannot be raised since there are no restrictions on file names.

The predefined package IO_EXCEPTIONS is defined as given in the *Ada Language Reference Manual [LRM]*.

Note that I/O operations on strings are implemented and operate in the normal way; it is only file I/O that is implemented as described above.

The predefined package LOW_LEVEL_IO is not implemented for the MC68020 target.

The implementation-dependent characteristics are described in Sections F.8.1 to F.8.4.

F.8.1 The Package SEQUENTIAL_IO

When any procedure is called, the exception STATUS_ERROR, MODE_ERROR or USE_ERROR is raised (there are no restrictions on the format of the NAME or FORM parameters).

F.8.2 The Package DIRECT_IO

When any procedure is called, the exception STATUS_ERROR, MODE_ERROR or USE_ERROR is raised (there are no restrictions on the format of the NAME or FORM parameters).

The type COUNT is defined:

```
type COUNT is range 0 .. INTEGER'LAST;
```

F.8.3 The Package TEXT_IO

When any procedure is called, the exception STATUS_ERROR, END_ERROR, MODE_ERROR or USE_ERROR is raised (there are no restrictions on the format of the NAME or FORM parameters). However, integer and real values can be read from, or written to, strings.

The type COUNT is defined:

```
type COUNT is range 0 .. INTEGER'LAST;
```

The subtype FIELD is defined:

```
subtype FIELD is INTEGER range 0 .. 255;
```

F.8.4 The Package IO_EXCEPTIONS

The specification of the package is the same as given in the *Ada Language Reference Manual [LRM]*.

F.9 Package STANDARD

The specification of package STANDARD is given in Figure F-2.

Figure F-2: Package STANDARD

```
package STANDARD is

   type BOOLEAN is (FALSE, TRUE);

   type SHORT_INTEGER is range
      - 128 .. 127;

   type INTEGER is range
      - 32_768 .. 32_767;

   type LONG_INTEGER is range
      - 2_147_483_648 .. 2_147_483_647;

   type FLOAT is digits 6 range
      - 16#0.FFFFFF#E32 .. 16#0.FFFFFF#E32;

   type LONG_FLOAT is digits 15 range
      - 16#0.FFFFFFFF_FFFFFFF#E44 ..
        16#0.FFFFFFFF_FFFFFFF#E44;


   type CHARACTER is
     (nul, soh, stx, etx,      eot, enq, ack, bel,
      bs , ht , lf , vt ,      ff , cr , so , si ,
      dle, dc1, dc2, dc3,      dc4, nak, syn, etb,
      can, em , sub, esc,      fs , gs , rs , us ,
      ' ', '!', '"', '#',      '$', '%', '&', ''',
      '(', ')', '*', '+',      ',', '-', '.', '/',
      '0', '1', '2', '3',      '4', '5', '6', '7',
      '8', '9', ':', ';',      '<', '=', '>', '?',
      '@', 'A', 'B', 'C',      'D', 'E', 'F', 'G',
      'H', 'I', 'J', 'K',      'L', 'M', 'N', 'O',
      'P', 'Q', 'R', 'S',      'T', 'U', 'V', 'W',
      'X', 'Y', 'Z', '[',      '\', ']', '^', '_',
      '`', 'a', 'b', 'c',      'd', 'e', 'f', 'g',
      'h', 'i', 'j', 'k',      'l', 'm', 'n', 'o',
      'p', 'q', 'r', 's',      't', 'u', 'v', 'w',
      'x', 'y', 'z', '{',      '|', '}', '~', del);
```

Figure F-2 Cont'd. on next page

Figure F-2 (Cont.): Package STANDARD

```
for CHARACTER use -- ASCII characters without holes
 (0  , 1  , 2  , 3  , 4  , 5  , 6  , 7  ,
  8  , 9  , 10 , 11 , 12 , 13 , 14 , 15 ,
  16 , 17 , 18 , 19 , 20 , 21 , 22 , 23 ,
  24 , 25 , 26 , 27 , 28 , 29 , 30 , 31 ,
  32 , 33 , 34 , 35 , 36 , 37 , 38 , 39 ,
  40 , 41 , 42 , 43 , 44 , 45 , 46 , 47 ,
  48 , 49 , 50 , 51 , 52 , 53 , 54 , 55 ,
  56 , 57 , 58 , 59 , 60 , 61 , 62 , 63 ,
  64 , 65 , 66 , 67 , 68 , 69 , 70 , 71 ,
  72 , 73 , 74 , 75 , 76 , 77 , 78 , 79 ,
  80 , 81 , 82 , 83 , 84 , 85 , 86 , 87 ,
  88 , 89 , 90 , 91 , 92 , 93 , 94 , 95 ,
  96 , 97 , 98 , 99 , 100, 101, 102, 103,
  104, 105, 106, 107, 108, 109, 110, 111,
  112, 113, 114, 115, 116, 117, 118, 119,
  120, 121, 122, 123, 124, 125, 126, 127);

package ASCII is

-- Control characters:

   NUL        : constant CHARACTER := nul;
   SOH        : constant CHARACTER := soh;
   STX        : constant CHARACTER := stx;
   ETX        : constant CHARACTER := etx;
   EOT        : constant CHARACTER := eot;
   ENQ        : constant CHARACTER := enq;
   ACK        : constant CHARACTER := ack;
   BEL        : constant CHARACTER := bel;
   BS         : constant CHARACTER := bs;
   HT         : constant CHARACTER := ht;
   LF         : constant CHARACTER := lf;
   VT         : constant CHARACTER := vt;
   FF         : constant CHARACTER := ff;
   CR         : constant CHARACTER := cr;
   SO         : constant CHARACTER := so;
   SI         : constant CHARACTER := si;
   DLE        : constant CHARACTER := dle;
   DC1        : constant CHARACTER := dc1;
   DC2        : constant CHARACTER := dc2;
   DC3        : constant CHARACTER := dc3;
   DC4        : constant CHARACTER := dc4;
   NAK        : constant CHARACTER := nak;
   SYN        : constant CHARACTER := syn;
   ETB        : constant CHARACTER := etb;
   CAN        : constant CHARACTER := can;
   EM         : constant CHARACTER := em;
   SUB        : constant CHARACTER := sub;
   ESC        : constant CHARACTER := esc;
   FS         : constant CHARACTER := fs;
   GS         : constant CHARACTER := gs;
   RS         : constant CHARACTER := rs;
   US         : constant CHARACTER := us;
   DEL        : constant CHARACTER := del;
```

Figure F-2 Cont'd. on next page

Figure F-2 (Cont.): Package STANDARD

```
-- Other characters:

   EXCLAM      : constant CHARACTER := '!';
   QUOTATION   : constant CHARACTER := '"';
   SHARP       : constant CHARACTER := '#';
   DOLLAR      : constant CHARACTER := '$';
   PERCENT     : constant CHARACTER := '%';
   AMPERSAND   : constant CHARACTER := '&';
   COLON       : constant CHARACTER := ':';
   SEMICOLON   : constant CHARACTER := ';';
   QUERY       : constant CHARACTER := '?';
   AT_SIGN     : constant CHARACTER := '@';
   L_BRACKET   : constant CHARACTER := '[';
   BACK_SLASH  : constant CHARACTER := '\';
   R_BRACKET   : constant CHARACTER := ']';
   CIRCUMFLEX  : constant CHARACTER := '^';
   UNDERLINE   : constant CHARACTER := '_';
   GRAVE       : constant CHARACTER := '`';
   L_BRACE     : constant CHARACTER := '{';
   BAR         : constant CHARACTER := '|';
   R_BRACE     : constant CHARACTER := '}';
   TILDE       : constant CHARACTER := '~';

-- Lower case letters:

   LC_A        : constant CHARACTER := 'a';
   LC_B        : constant CHARACTER := 'b';
   LC_C        : constant CHARACTER := 'c';
   LC_D        : constant CHARACTER := 'd';
   LC_E        : constant CHARACTER := 'e';
   LC_F        : constant CHARACTER := 'f';
   LC_G        : constant CHARACTER := 'g';
   LC_H        : constant CHARACTER := 'h';
   LC_I        : constant CHARACTER := 'i';
   LC_J        : constant CHARACTER := 'j';
   LC_K        : constant CHARACTER := 'k';
   LC_L        : constant CHARACTER := 'l';
   LC_M        : constant CHARACTER := 'm';
   LC_N        : constant CHARACTER := 'n';
   LC_O        : constant CHARACTER := 'o';
   LC_P        : constant CHARACTER := 'p';
   LC_Q        : constant CHARACTER := 'q';
   LC_R        : constant CHARACTER := 'r';
   LC_S        : constant CHARACTER := 's';
   LC_T        : constant CHARACTER := 't';
   LC_U        : constant CHARACTER := 'u';
   LC_V        : constant CHARACTER := 'v';
   LC_W        : constant CHARACTER := 'w';
   LC_X        : constant CHARACTER := 'x';
   LC_Y        : constant CHARACTER := 'y';
   LC_Z        : constant CHARACTER := 'z';

end ASCII;
```

Figure F-2 Cont'd. on next page

Figure F-2 (Cont.): Package STANDARD

```
   -- Predefined subtypes:

   subtype NATURAL  is INTEGER range 0 .. INTEGER'LAST;

   subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;

   -- Predefined string type:

   type STRING is array (POSITIVE range <>) of CHARACTER;

   type DURATION is delta 2#1.0#E-7 range
      - 16_777_216.0 .. 16_777_215.0;

   -- The predefined exceptions:

   CONSTRAINT_ERROR : exception;
   NUMERIC_ERROR    : exception;
   PROGRAM_ERROR    : exception;
   STORAGE_ERROR    : exception;
   TASKING_ERROR    : exception;

end STANDARD;
```

F.10 Package MACHINE_CODE

Package MACHINE_CODE is not supported by this version of the compiler.

F.11 Language-Defined Pragmas

The definition of certain language-defined pragmas is incomplete in the *Ada Language Reference Manual [LRM]*. The implementation restrictions imposed on the use of such pragmas are specified in Sections Section F.11.1 to Section F.11.5.

F.11.1 Pragma INLINE

This pragma supplies a recommendation for inline expansion of a subprogram to the compiler. This pragma is ignored by this version of the compiler.

F.11.2 Pragma INTERFACE

This pragma allows subprograms written in another language to be called from Ada. The compiler only supports pragma INTERFACE for ASSEMBLER and RTS.

F.11.2.1 Assembler

Normal Ada calling conventions are used by the compiler when generating a call to an ASSEMBLER subprogram. The calling mechanism is described in Section F.11.2.1.3. Further information is given in the *Target Handbook {TH}*.

F.11.2.1.1 Assembler Names

The name of an interface routine must conform to the naming conventions both of Ada and of the Ada-Plus MC68020 Assembler. If the Ada identifier is longer than 12 characters then the interface name is the Ada identifier truncated to 12 characters. Underscore (_) characters in Ada subprogram names are translated to dollar ($) characters in the call of the assembly code subprogram. The user must therefore follow this convention when writing the assembly code body. The rest of the characters are restricted to being underlines, digits or letters. Names within assembler code must use uppercase letters.

F.11.2.1.2 Parameter-Passing Conventions

Parameters are passed to the called procedure in the order given in the specification of the subprogram, with default expressions evaluated, if present.

Scalars are passed by copy for all parameter *modes* (the value is copied out for parameters with mode out).

Composite types are passed by reference for all parameter modes.

F.11.2.1.3 Procedure-Calling Mechanism

Normal Ada calling conventions are used by the compiler when generating a subprogram call.

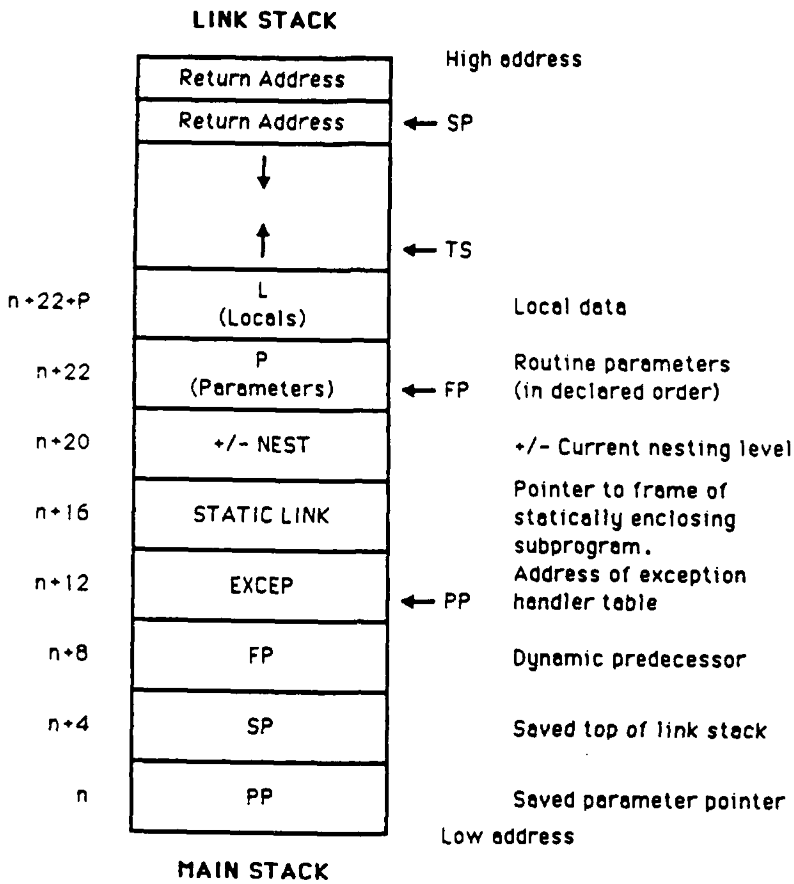
The procedure-calling mechanism uses the run-time stack organisation shown in Figure F-3, and the routine entry and exit code shown in Figure F-4. Note that the return link is maintained automatically on a separate stack (SP).

The implementation uses the following dedicated and temporary registers:

SP	-	Link Stack Pointer	A7
FP	-	Frame Pointer	A2
PP	-	Parameter Frame Pointer	A3
TB	-	Task Base Pointer	A0
TS	-	Main Stack Pointer	A1

These values must be preserved by any assembler code.

Figure F-3: Routine Activation Record on Entry to Called Subprogram



Macros RM_P_BEGIN and RM_P_END are provided in the macro library contained within the program library for the routine entry and exit code respectively. This code is shown in Figure F-4.

Figure F-4: Routine Entry And Exit Code

Routine Entry Code :

```
MOVE.L      SP,(PP)+
MOVE.L      FP,(PP)+
MOVEA.L     PP,FP
MOVE.L      FP,(FP)+
MOVE.L      A4,(FP)+
if frame_is_indirect then     * Frame should never be indirect for this macro.
MOVE.W      #-<nest>,(FP)+
else
MOVE.W      #<nest>,(FP)+
end if
```

Routine Exit Code :

```
MOVEA.L     -(PP),FP
MOVEA.L     -(PP),SP
RTS
```

F.11.2.2 RTS

RTS provides a more efficient calling mechanism, although restrictions are placed on the number of parameters by the number of available registers. The primary purpose of RTS is for run-time system calls.

F.11.2.2.1 RTS Names

(see Section F.11.2.1.1, Assembler Names)

F.11.2.2.2 Parameter-Passing Conventions

(see Section F.11.2.1.2, Parameter-Passing Conventions).

The parameters, P1 .. Pn, are passed in the corresponding order in data registers, D0 .. Dn-1. If the parameters are floating point types, they are passed in floating point registers FP0 .. FPn-1. The number of parameters, n, is restricted to six.

F.11.2.2.3 Procedure-Calling Mechanism

Procedures are called directly, no entry and exit code macros are necessary. The procedure-calling mechanism is outlined in the following example:

```
procedure P is (X : in INTEGER; Y : out INTEGER);

pragma INTERFACE (RTS, P);
```

X is passed in D0, Y is passed back in D1.

F.11.3 Pragma OPTIMIZE

This pragma supplies a recommendation to the compiler for the criterion upon which optimisation is to be performed. This pragma is ignored by this version of the compiler.

F.11.4 Pragma PACK

Form

pragma PACK(*type*_simple_name);

Takes the simple name of record or array type as a single argument.

Position

The allowed positions for this pragma, and the restrictions on the named type, are governed by the same rules as for a representation clause; in particular, the pragma must appear before any use of representation attribute of the packed entity.

Effect

The pragma specifies that storage minimization to storage unit boundary is the main criterion when selecting the representation of the given type.

F.11.5 Pragma SUPPRESS

This pragma gives permission for specified run-time checks to be omitted by the compiler. This pragma is ignored by this version of the compiler.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name_and_Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	AA....A1 |-----| 511 characters
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	AA....A2 |-----| 511 characters
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	AA....A3A....A |-----| |----| 255 256 characters
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	A....A4A....A |----| |----| 255 256 characters
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	0....0298 |----| 509 characters
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	0....069.0E1 |----| 506 characters
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	A....A |----| 256 characters

Name_and_Meaning	Value
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	A....A1 |----| 255 characters
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	492 blanks
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	32767
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	X}]!.dat
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	file*.dat
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	2.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	16777216.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	bad_char^

Name_and_Meaning	Value
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	NO_SUCH_NAME_POSSIBLE
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32768
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32767
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32768
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-3.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-16777216.0
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	512
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	0....02:11: |----| 507 characters

Name_and_Meaning	Value
\$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	0....016:F.E: |----| 505 characters
\$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	"A....A" |----| 510 characters
\$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
\$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE
\$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform the Ada Standard. The following 27 tests had been withdrawn at time of validation testing for the reasons indicated. A reference the form "AI-ddddd" is to an Ada Commentary.

B28003A: A basic declaration (line 36) wrongly follows a l declaration.

E28005C: This test requires that 'PRAGMA LIST (ON);' not appear i listing that has been suspended by a previous "pragma (OFF);"; the Ada Standard is not clear on this point, and matter will be reviewed by the ALMP.

C34004A: The expression in line 168 wrongly yields a value outside the range of the target T, raising CONSTRAINT_ERROR.

C35502P: The equality operators in lines 62 and 69 should inequality operators.

A35902C: Line 17's assignment of the nominal upper bound of a f point type to an object of that type raises CONSTRAINT_E for that value lies outside of the actual range of the ty

C35904A: The elaboration of the fixed-point subtype on line wrongly raises CONSTRAINT_ERROR, because its upper b exceeds that of the type.

C35904B: The subtype declaration that is expected to ra CONSTRAINT_ERROR when its compatibility is checked aga that of various types passed as actual generic paramet may in fact raise NUMERIC_ERROR or CONSTRAINT_ERROR reasons not anticipated by the test.

C35A03E: This test assumes that attribute 'MANTISSA' returns 0 applied to a fixed-point type with a null range, but the Standard doesn't support this assumption.

C35A03R: This test assumes that attribute 'MANTISSA' returns 0 applied to a fixed-point type with a null range, but the Standard doesn't support this assumption.

C37213H: The subtype declaration of SCONS in line 100 is wro expected to raise an exception when elaborated.

C37213J: The aggregate in line 451 wrongly raises CONSTRAINT_ERROR

C37215C: C37215E: C37215G: C37215H: Various discriminant constraints are wrongly expected to incompatible with type CONS.

C38102C: The fixed-point conversion on line 3 wrongly ra CONSTRAINT_ERROR.

C41402A: 'STORAGE_SIZE' is wrongly applied to an object of an ac type.

C45332A: The test expects that either an expression in line 52 raise an exception or else MACHINE_OVERFLOWS is FA Howeve~, an implementation may evaluate the expres correctly using a type with a wider range than the base of the operands, and MACHINE_OVERFLOWS may still be TRUE.

C45614C: REPORT_IDENT_INT has an argument of the wrong (LONG_INTEGER).

A74016C: C85018B: C87B04B: CC1311B: A bound specified in a fixed-point subtype declaration ϕutside that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur re lines 37 59, 142 and 143, 16 and 48, 252 and 253 of the four t respectively (and possibly elsewhere).

BC3105A: Lines 159..168 are wrongly expected to be incorrect; they correct.

AD1A01A: The declaration of subtype INT3 raises CONSTRAINT_ERROR implementations that select INT'SIZE to be 16 or greater.

CE2401H: The record aggregates in lines 105 and 117 contain the w values.

CE3208A: This test expects that an attempt to open the default ou file (after it was closed) with mode IN_FILE ra NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_E should be raised.